Simulating the Sequoia AVC Advantage DRE Voting Machine

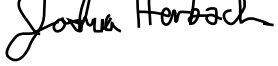
Joshua S. Herbach
Department of Computer Science
Princeton University
herbach@cs.princeton.edu

May 2007

A Computer Science BSE Independent Work Project

Prof. Andrew W. Appel, *Advisor*

This paper is my own work in accordance with University Regulations.



Abstract

Recent research has demonstrated that some of the most commonly used paperless Direct-Recording Electronic Voting Machines are vulnerable to attacks which threaten the accuracy and security of elections in which they are used. This paper is one part of an independent security study analyzing the Sequoia AVC Advantage DRE voting machine which was the second most commonly used DRE in the November 2006 general elections. To aid in our analysis of the Advantage, we implemented a combined simulator and debugger using open-source libraries and tools. To prevent leakage of copyrighted intellectual property into the simulator, we maintained a "Chinese Wall" between the team examining the Advantage and the team which implemented the simulator. The resulting simulator has proven useful for analyzing the Advantage's firmware. As our study proceeds, we expect it to both help improve our understanding of the Advantage's hardware and aid in the development of working demonstrations of attacks.

1 Introduction

An idealized democracy requires fair elections – elections in which individuals cannot be coerced or bribed into voting against their will, and in which each individual's vote is correctly counted only once. The reality is that elections are not necessarily fair. For example, during the Gilded Age political bosses regularly bought and sold votes [10]. Similarly, attacks on election systems have been the basis of many fictionalized accounts of election fraud [9][16][13]. Designing a fair voting mechanism for elections is notoriously difficult and is an open research problem [15][2]. Even with over two hundred and fifty years (starting this reckoning during colonial times [4]) to create a secure voting system,[1] election fraud still remains an issue of concern in the United States [11].

During the November 2006 general election, 36% of the counties (containing a total of 38% of the registered population) [5] in the United States used Direct-Recording Electronic (DRE) voting machines, like the Sequoia AVC Advantage (Advantage). The Advantage was used in counties containing a total of 6.6% of all registered voters, and was the second most commonly used DRE in the 2006 election.[2] While the large-scale introduction of DRE voting machines does not necessarily compromise an already fair election system, it does create many new opportunities for election fraud. Fundamentally, DRE voting machines are computers – much like regular desktop PCs – with some specialized hardware and a software program[3] designed to count votes. Like most general-purpose computers, the program running on DREs such as the Advantage can be changed and any modifications could cause a DRE to miscount votes. The Advantage's processor closely resembles that of the original Nintendo Game Boy. Anyone knowledgeable enough to have done development work on games for the original Game Boy[4] could conceivably write a program to (mis)count votes. Modifications to the program running on the Advantage do not need to take place at the factory; like a Game Boy it is possible to remove and insert a device containing the modified program to be run. The program running in an Advantage is stored in a set of chips. A determined individual could access the chips merely by picking a lock (this has been demonstrated to take as little as seven seconds) and removing ten screws [1]. Replacing the firmware chips with chips containing a modified program requires no special tools – a screwdriver could be used to remove the chips and new chips could be inserted by hand.

Figure 1 Sequoia AVC Advantage

The Advantage does not generate a voter-verified paper ballot, and, as a result, there is no way for a voter to confirm that the machine counted his vote correctly – a machine could display one selection on its display, but instead record a different selection without the voter's knowledge. Similarly, no recount can take place. An incorrect count could be a result of a malicious

[1] Assuming, of course, a secure voting system is and has always been desirable in the United States.
[2] The first most commonly used DRE, the Diebold AccuVote-TS was used by about 9.5% of all registered voters and already has been the subject of a security study [6].
[3] Hereafter, software program, software, firmware, and program will all be used interchangeably
[4] Such as any first-year computer science undergraduate

party replacing the software in the DRE as described above, or, alternatively, it could be caused by a bug in the software provided by Sequoia. Decades of work with computer systems has shown that detecting bugs in software – much less ensuring said bugs are fixed and the software is secure – is difficult at best. Consequently, assuming the original Advantage software will always work perfectly is fallacious.

The purpose and result of this project was to create a combination simulator and debugger for the Sequoia AVC Advantage DRE voting machine. This simulator and debugger is portable and is accessible to anyone interested in studying the system without violating the legal rights of the Advantage's creators. With the simulator and debugger it is now possible to

- Run the Advantage's original firmware in the simulator, and pause its execution at any point to examine the internal state of the machine
- Develop and test new firmware for the Advantage without the delay of clearing old data from a chip, burning the new firmware onto it, and then installing the chip into an Advantage
- Test firmware and possible attack vectors on the Advantage without requiring physical access to one of the machines

Even though we are still in the process of reverse-engineering the Advantage, we have already found the simulator to be a useful aid in the study of the Advantage's firmware and as the reverse-engineering process continues, we only expect the simulator to become a more effective tool.

2 Reverse Engineering and Intellectual Property

To simulate the Advantage, we first had to determine the system's behavior. One option was to systematically try every single possible input to the system and carefully record the output (a crude form of reverse engineering). A more practical approach was to examine the system's hardware and firmware to determine the interface between the firmware and the hardware and the behavior of the hardware, and to then model the simulator based on these findings. There are multiple possible approaches to reverse-engineering a system like the Advantage. One approach is purely hardware-based: trace every single wire on every circuit board and identify all of the pins of all of the components that are attached in a net and then use this information to determine the system's behavior. Another approach also examines the firmware and identifies how the processor interacts with the various peripheral components and uses this information in conjunction with information gleaned from examining the hardware.

Examining the hardware or firmware and then using the knowledge gained through said examination to make the simulator requires consideration of some legal issues. If the hardware design of the system was patented,[5] that protection would last for twenty years. The firmware in all of its forms (e.g. source code, compiled binaries, and disassembly of the binaries) is protected under copyright law for many more years – the lifetime of author plus seventy or ninety-five years. Because the Advantage was first introduced in 1987, patents impose no legal restrictions on this project because any patents protecting its hardware have now (2007) expired. However, any simulator development that takes place with access to the firmware could arguably mean that Sequoia's copyrighted intellectual property has "leaked" into the simulator. While developing and using the simulator for research purposes would be acceptable "fair use" in accordance with legal requirements, distributing the simulator to anyone who wanted to use it could violate copyright laws.

[5] It is unclear whether the Advantage was ever patented. At about the same point in time as the introduction of the Advantage, similar systems were patented [3]; however, what, if any, relation there is between these systems and the Advantage is ambiguous.

To preclude any intellectual property issues, our team divided into two groups and adopted a Chinese Wall policy. One group worked with the Advantage, analyzing the hardware and firmware and writing functional specifications. The other group worked only on the simulator. The only contact between the two groups was through an archived mailing list, onto which the simulator team posted questions, and the reverse-engineering team posted functional specifications and responses to those questions. This effectively isolated the simulator development team from knowledge of the firmware and the archived mailing list could serve as proof that no knowledge of the voting machine firmware was passed on to the simulator team.

While this is a sound approach in terms of compliance with legal constraints, it hindered simulator development as it restricted the resources we could allocate to analyzing the Advantage's firmware and imposed a communication barrier between those with knowledge about the inner workings of the Advantage, and those whose work was based on that knowledge. As a result, the simulator group's progress was limited by the progress of the reverse-engineering group, and when the reverse-engineering group fell behind the simulator group, it was not possible to temporarily shift workers from the simulator group to the reverse-engineering group to address the problem.

3 Advantage Architecture Overview

At the core of the Advantage is an 8-bit Zilog Z80 microprocessor. The software executed by the Advantage is stored on a set of three 64 KB[6] ROM chips and a 32 KB RAM chip. The Advantage also has a series of RAM chips[7] on which it stores and manipulates data. To access this program and data memory, the Advantage uses the Z80's memory bus. The Z80 has a 16-bit address bus and has instructions for reading and writing individual bytes or groups of bytes from external memory based on the address placed on the address bus. However, it is only possible to address 64 KB directly. To access all of the memory in the program and data ROM and RAM chips, the Advantage's Z80 core has two peripheral subsystems dedicated to memory management: one manages the program memory which is accessed using the lower 32 KB of the address-space; the other manages the data memory, which is accessed through the upper 32 KB of the address-space [7]. The Z80 communicates with its other peripherals through two mechanisms: peripherals that can trigger processor interrupts, and the Z80's set of port IO instructions.[8] These IO instructions allow for byte-by-byte reading and writing to and from external peripherals addressed using one of 256 ports [17].[9] The memory manager subsystems are good examples of how the Z80 communicates with its peripherals. To control the program and data memory managers, the firmware executes an **`out`** instruction on the processor, writing a byte to port 1 or 2 to select a mapping from 16-bit addresses to physical chips [7].

All of the specialized hardware of the Advantage is accessed by the firmware using port IO instructions. The peripherals that have been reverse-engineered at the time of this report's writing include [7]:

---

[6] Some models of the Advantage may contain 128 KB ROM chips.

[7] The Advantage contains a 32 KB RAM chip and one or two 128 KB RAM chips; the number of 128 KB RAM chips can vary from machine to machine.

[8] The two basic port IO instructions are **`in`** and **`out`** which – respectively – read a byte from, or write a byte to a port.

[9] There is a slight caveat to this: while officially a port is specified by a single byte, giving a total of 256 easily addressable peripherals, that byte plus another byte are placed on the 16-bit address bus. As a result, if the peripheral hardware is designed appropriately, it is possible to easily address 65536 external peripherals. Both the one and two-byte peripheral addressing schemes are used by the Advantage. It is also possible to extend the number of addressable peripherals by using the first byte written to a port to determine where to direct subsequent port operations – this mechanism is the basis for how the program and data memory managers work.

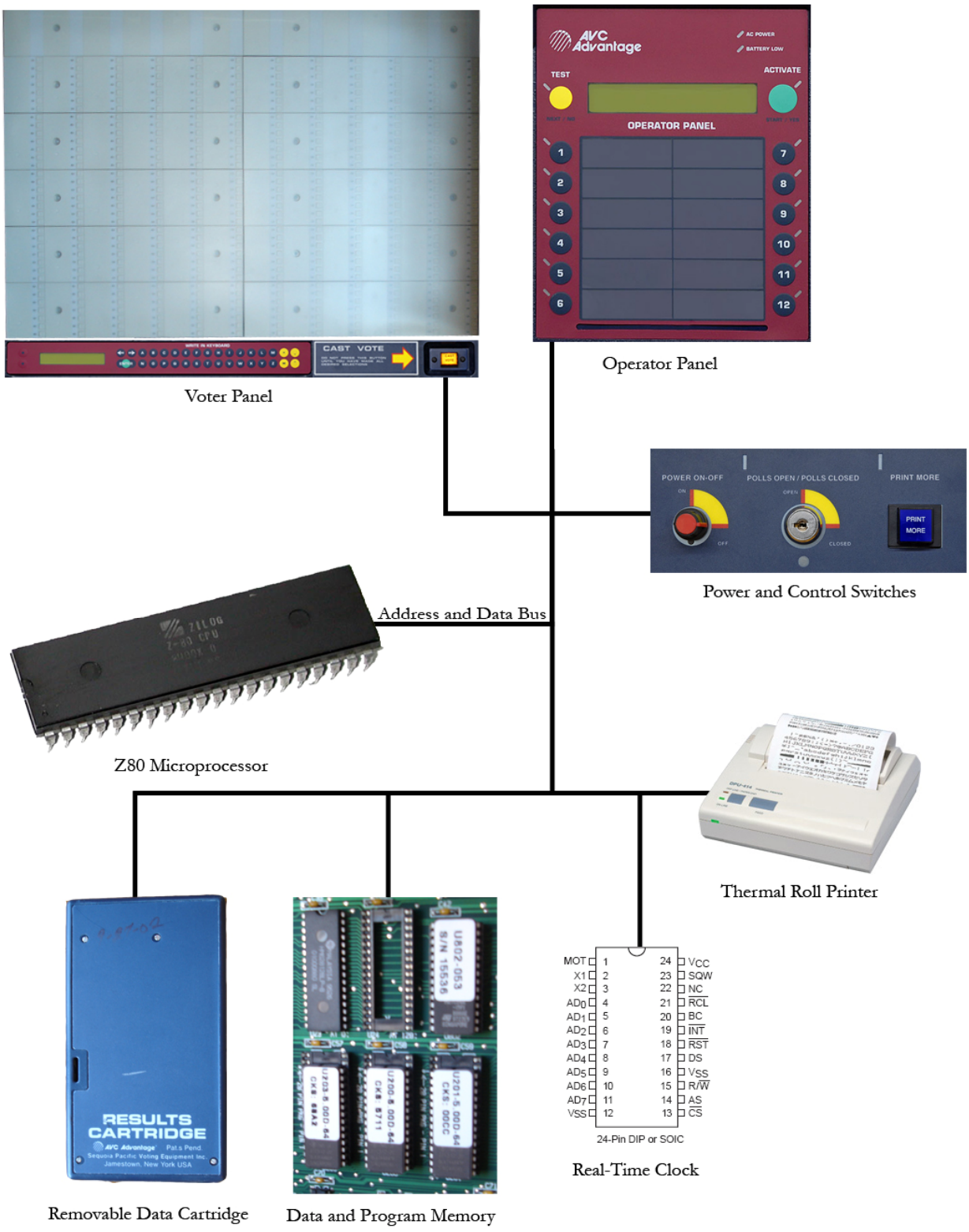


Figure 2 Diagram illustrating the connection between the Z80 processor and the reverse-engineered peripherals that have been integrated into the simulator

- Results Cartridge
  - A removable memory cartridge used to hold ballot definitions and results
- Operator Panel
  - The LCD screen, LEDs and buttons presented to poll workers controlling the Advantage
- Configuration ROM
  - An 8 KB ROM chip (256 B are accessible) to hold the Advantage's serial number and cryptographic key
- Additional memory
  - An 8 KB RAM chip
- Real-time clock
  - A TI BQ3285 chip which supports the generation of square waves, periodic interrupts, alarms and both normal binary and BCD encoded output
- Thermal roll printer
  - A Seiko DPU-414 accessed through a standard 25-pin parallel port
- Power control
  - A mechanism for controlling the powering of the Advantage and its various peripherals and for reading the status of a set of control switches presented to the operator
- Voter Panel
  - The LCD screen, LEDs and buttons presented to voters using the Advantage
- Interrupt Controller
  - A pair of external Z84C30 counter/timer circuits control the maskable interrupts to the processor; non-maskable interrupts can be raised when an op code is fetched from RAM in the data section, or when a watchdog bit is not modified
- Voltage Monitor
  - ADC used to monitor the voltage levels of various system batteries

A few non-essential peripherals including the sound generator are still being reverse-engineered.

4 Implementing the Simulator

Because the Z80 microprocessor in the Advantage runs at 4 MHz and it is desirable for the simulator to run in real-time, it is important that the emulator of the Z80 and the entire simulator run quickly. An additional requirement for the simulator is that it should be portable and not restricted to a single platform. To meet both of these requirements, we chose to use C to implement the debugger and simulator internals and Java to implement a portable front-end.

4.1 Z80 Emulation

There already exist many Z80 emulators written in C – they are at the core of emulators for the Nintendo Game Boy [14] and other systems. Rather than writing an emulator from scratch, we based our Z80 emulator on Lorenzo Lucchini's open source Z80Sim [12]. The main advantages of the Z80Sim over other Z80 emulators for this project are that it is open source – it can be freely modified and redistributed – the processor code is relatively clean and straightforward, and there is a fairly simple interface for extending and modifying it. There also are many shortcomings to using the Z80Sim: it does not implement some of the Z80 op codes, its timing information for some instructions do not match the data published by Zilog, it does not support most of the Z80 interrupt modes, a few of its op codes do not work properly, it does not correctly maintain a number of the

Z80s flags, it operates on the assumption that only 64 KB of memory would ever be addressed by the Z80, and finally, it does not support port IO instructions where peripherals are selected using the full 16-bits of the address bus – not just the single byte specified in the Z80 manual.

Our first step in working on the Advantage simulator was to address the aforementioned shortcomings of Z80Sim. We added missing state registers and fixed existing op codes, and then moved on to implementing the missing op codes and defining a new interface for memory accesses. Also, from a software engineering perspective, the Z80Sim lacked cleanly defined interfaces and modularity and had a fairly polluted global namespace. As a result, part of setting up the Z80 emulator involved cleaning up the Z80Sim code – i.e. restricting the scope of many variables and functions (making many of them static), and imposing a consistent naming scheme on the functions the Z80 emulator would expose to other modules.

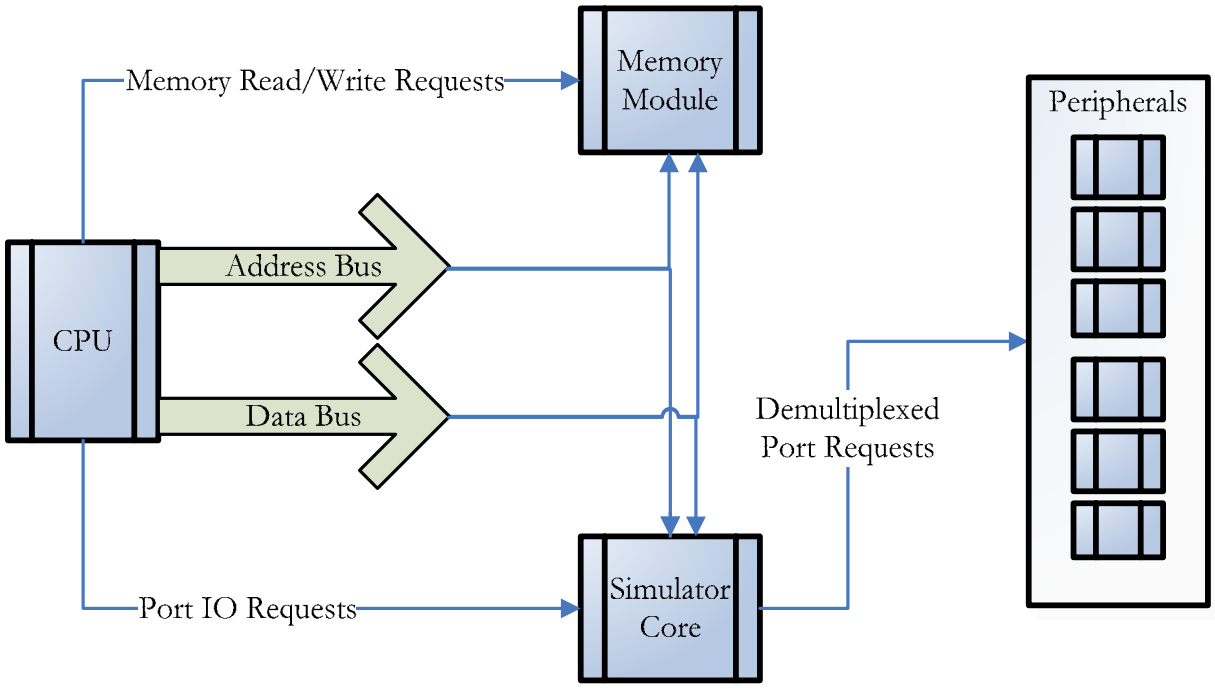


Figure 3 Basic simulator architecture, ignoring interrupts and peripherals requiring user interaction and timing information

The final Z80 emulator is split into three main components: (1) the CPU module – which contains the state of the processor and the encoding for how instructions are to be executed, (2) a memory module – which controls the mapping from addresses to storage bytes (initially this mapping was 1:1; with the implementation of the memory controller peripherals, it becomes significantly more complex), and (3) the simulator core – which is responsible for the initialization of the simulator, the control of when the processor steps, and the interface between the processor and all other peripherals (see section 4.5 for an expanded description of the simulator core).

4.2 Peripheral Emulation

We defined the following basic interface for integrating modules that implement peripherals into the simulator:

Table 1. Basic peripheral interface

Function	*Description*
`Module_Initialize(ConfigData)`	Initializes the peripheral module using the specified configuration data
`Module_Finalize(ConfigData)`	Performs any necessary finalization on the peripheral module
`Module_In(Port)`	Models the behavior of the peripheral when a port input instruction is executed on the specified port; returns the byte the **`in`** instruction reads
`Module_Out(Port, Data)`	Models the behavior of the peripheral when an port output instruction is executed on the specified port with the specified data

Because most of the Advantage's peripherals have their state in a well-defined form at start-up, an initialization step must be performed on most of the peripherals' modules. Some modules (e.g. the memory module) have initial state which can vary from machine to machine. An example of this is the fact that each machine has a different set of battery-backed RAM chips and thus each machine has different data stored in its RAM when activated. Rather than hard-coding these parameters, we introduced a configuration file to specify all of the parameters – appendix A.2 describes the file format. Similarly, when the Advantage is shut down, some peripheral's final state is retained (for example, the data memory subsystem consists of battery-backed RAM, so when the machine is shut down, data in the RAM is maintained in the RAM) and so a final cleanup operation is necessary for some modules. Further, because basic access to all peripherals takes place through the Z80's port IO instructions, almost all peripherals need to implement functions to handle port **`in`** and **`out`** instructions (because many peripherals are accessed using multiple ports, inclusion of the port in the function is necessary for determining what operation the peripheral module should perform).

Although this basic interface suffices to completely model the configuration ROM and the extra 8 KB RAM chip, most peripherals require one (or more) additional interfaces for handling timing, user-interaction, or additional side-channel communication.

4.2.1 Peripheral Timing

A number of peripherals have state which is dependant upon timing. For example, the LCDs used in the operator panel and in the write-in portion of the voter panel require a short delay between each instruction issued to them. Rather than using the system clock of the computer running the simulator to model this delay, the simulator instead uses the clock of the Z80 emulator – when a command is issued to an LCD through a port output command, the current CPU time is recorded along with the required delay. Then, when a new command is sent to the LCD, the LCD checks the current CPU time and determines whether or not the full delay has elapsed and then can determine what action to take. Choosing to use the emulated processor's time instead of the system clock of the computer is logical because it more closely models the behavior of the Advantage in normal operation. Additionally, many of the peripherals need timing information with sub-millisecond accuracy, and because the slowest Z80 instruction takes fewer than 30 clock cycles, the granularity of the emulator clock is on the order of at most 7.5µs.[10]

[10] Most desktop computer systems do not provide sub-millisecond timing information.

In general, peripherals' need for timing information takes two forms: (1) some peripherals, like the LCDs, only need to know timing information when an operation is performed on them and these peripherals can be most efficiently modeled by polling the clock when they perform operations; (2) other peripherals need timing information at regular intervals – the easiest way to model this is to constantly update these peripherals with timing information so they can choose when they need to perform an action (an example of such a device is the voter panel LEDs which, when not driven, must remain lit for about .16s before deactivating). To accommodate both of these needs, we defined pushing and polling interfaces for timing information.

Table 2. Peripheral timing interface

Function	*Description*
`Processor_GetTStates()`	Polls the processor for current timing information; it returns the lower 4 bytes of the number of elapsed clock cycles
`Processor_GetTStatesOverflow()`	Polls the processor for current timing information; it returns the upper 4 bytes of the number of elapsed clock cycles
`Module_Tick(time)`	After each Z80 instruction is executed, the simulator core executes all _Tick functions, pushing out the current elapsed number of clock cycles

4.2.2 User Interaction and Peripherals

Many peripherals have state which is either set by user actions (e.g. pushing buttons) or has to be displayed to the user (e.g. the activation of LEDs, lights, or LCDs). We defined yet another interface so that these peripherals could be used in the simulator.

Table 3. Peripheral user interaction interface

Function	*Description*
`Module_Print()`	Output a peripheral's displayable state – LEDs, LCDs, etc – in ascii (for use with the debugger)
`Module_PrintState()`	Output the internal state of a peripheral – registers and status bits not normally externally visible (for use with the debugger)
`Module_ToggleButton(button)`	Allows the user to (in the debugger) toggle a button
`Module_UpdateFrontEnd()`	Causes a module to dump any changes to its state to the front-end to be displayed
`Module_ProcessMessage(message)`	Causes a module to process a message from the front end specifying user input

4.2.3 Peripherals Requiring Additional Communication Channels

The final set of functions required to model peripherals is less well defined than the above interfaces. Peripherals such as the data and program memory managers must interact with the memory module that controls memory reads and writes by the processor – no message is passed between the memory managers and the memory module using any of the standard interfaces (no port IO instruction takes place, and it does not involve timing or user interaction). Similarly, the power control peripheral must send information to the voter panel peripheral specifying whether the voting

panel electronics even have any power; this information does not fit under any of the standard interfaces. No single interface addresses these requirements and so we adopted an ad hoc approach to handle these requirements. In the case of the memory managers, both memory manager subsystems were incorporated directly into the memory module, thus eliminating the need for an additional channel and an associated set of interface functions. For the power control peripheral, the easiest solution was to introduce an additional set of functions for specifying whether or not the voter panel is powered.

4.3 Front-End

We implemented the simulator's front-end in Java using the Swing and AWT libraries. It provides a GUI for users to view representations of the operator and voter panels that are exposed to users of the Advantage (see figure 4 for an example of the GUI).

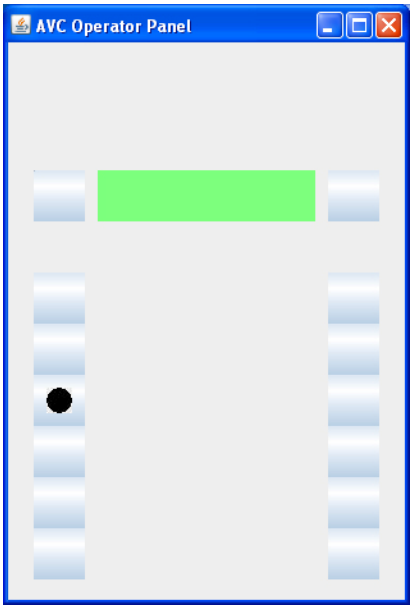


Figure 4 Operator Panel currently presented by the simulator

Rather than using the JNI for interfacing the GUI and the simulator, we chose to use sockets for inter-process communication between the Java front-end and the C back-end. Because of cross-platform issues with socket libraries,[11] the simulator uses an adaptation of the open source C util_network library created for and used in ChucK.[12] Messages of the form defined in Table 4 are used to pass information to the front-end about which LEDs and lights should be activated and about what text should be displayed on the LCDs, and to pass information back to the simulator about which buttons have been pushed.

In the Java front-end, communication between the simulator and the GUI is done in a separate thread. In the simulator, communication with the front-end is not done in a separate thread, but instead takes place in the main loop of the core of the simulator (see section 4.5). After executing each instruction, all of the **`UpdateFrontEnd`** functions specified in the interface of 4.2.2 are executed – accumulating messages for the front-end – and then those messages are delivered to the front-end. Similarly, after executing each instruction, the simulator core polls the front-end for any update messages, and the front-end responds by sending back messages describing any user inputs, which are then passed on to the appropriate **`ProcessMessage`** function (again specified in the interface of 4.2.2).

Table 4. Structure of messages passed between the front-end and simulator

Field	*Description*
`Component`	Value used to determine which component of the front-end or simulator the message should be passed to
`Length`	Length of data packet
`Data` (length bytes)	Definition varies between components

[11] In particular, Windows' socket libraries require additional WSA function calls in order to perform basic socket operations.

[12] A Strongly-timed, On-the-fly Audio Programming language – http://chuck.cs.princeton.edu

This implementation of the front-end is fairly straightforward, but has issues in terms of the timing of user input. The Java Swing library does not provide a simple mechanism for knowing when, or for how long, a button has been held down; instead, it just reports when a button has been pushed and released. Because the Advantage's mechanism for reading user input is to set a bit when a button is pushed and to clear that bit when the button is released, there is an incompatibility between Swing's basic event model, and the requirements of accurately simulating the Advantage. To work around this problem, the front-end records when a button is pressed and begins a timer based on clock information sent by the simulator core (this clock information is again taken from the Z80 emulator). The front-end considers a button held for a short time period following Swing raising the event indicating it was triggered. This solution introduces some idiosyncrasies when the simulator is run in debug mode: pushing a button at one instant means that it will be considered pushed for a series of subsequent instructions which – if an investigator is stepping through code very slowly – might take hours. This solution, and really any solution using the basic events of the Swing library, also makes it difficult to investigate the role of debouncing in the Advantage – it is entirely possible that any debouncing that is taking place in the Advantage's software could generate erroneous data.

4.4 Debugger

While Z80Sim did come with a debugger that supported watch points (and therefore also breakpoints) it was written in such a fashion that it was hard, if at all possible, to reasonably extend it. Because of this difficulty, and the fact that many of the features supported by the Z80Sim debugger were not useful in analyzing unknown software,[13] we chose to write our own debugger.

Our debugger consists of two layers: the front-end and an interpreter. The front-end performs lexical analysis of the user input and then generates a parse tree. The lexer and parser are implemented using lex and yacc – flex and bison for the Window's build. The front-end is designed to be easily extended; just a few lines of additional code are required to specify how

```
Version: 0.007
Z80 initialized
Operator Panel initialized
Voter Panel LCD initialized
Voter Panel initialized
Attempting to establish connection with FrontEnd Client
        If the FrontEnd application isn't already started, please start it (java
 -jar AVCGui.jar)
Connected to FrontEnd!
(avc) step
CALL 0x13
(avc) break 0x59
Break: b0 V:0,59
(avc) c
breaking b0 V:0,59
OUT (0x12), A
(avc) s
RET
(avc) clear 0x59
Can only clear an already existing breakpoint
(avc) clear b0
Disable Break: V:0,59
(avc) continue
PC: 26 (P:0,26); read port (unrecognized): 0 loaded 0x48 from input script
PC: 28 (P:0,28); write port (unrecognized): 0 data: 48
PC: 2a (P:0,2a); read port (unrecognized): 0 loaded 0x45 from input script
PC: 2c (P:0,2c); write port (unrecognized): 0 data: 45
PC: 2e (P:0,2e); read port (unrecognized): 0 loaded 0x4c from input script
PC: 30 (P:0,30); write port (unrecognized): 0 data: 4c
PC: 32 (P:0,32); read port (unrecognized): 0 loaded 0x4c from input script
PC: 34 (P:0,34); write port (unrecognized): 0 data: 4c
PC: 36 (P:0,36); read port (unrecognized): 0 loaded 0x4f from input script
PC: 38 (P:0,38); write port (unrecognized): 0 data: 4f
PC: 3a (P:0,3a); read port (unrecognized): 0 loaded 0x20 from input script
PC: 3c (P:0,3c); write port (unrecognized): 0 data: 20
PC: 3e (P:0,3e); read port (unrecognized): 0 loaded 0x57 from input script
PC: 40 (P:0,40); write port (unrecognized): 0 data: 57
PC: 42 (P:0,42); read port (unrecognized): 0 loaded 0x4f from input script
PC: 44 (P:0,44); write port (unrecognized): 0 data: 4f
PC: 46 (P:0,46); read port (unrecognized): 0 loaded 0x52 from input script
PC: 48 (P:0,48); write port (unrecognized): 0 data: 52
PC: 4a (P:0,4a); read port (unrecognized): 0 loaded 0x4c from input script
PC: 4c (P:0,4c); write port (unrecognized): 0 data: 4c
PC: 4e (P:0,4e); read port (unrecognized): 0 loaded 0x44 from input script
PC: 50 (P:0,50); write port (unrecognized): 0 data: 44
User Break
(avc) printop
AC Power *
Battery Low
Test     Activate
HELLO WORLD

 1 *      7
 2        8
 3        9
 4       10
 5       11
 6       12
(avc)
```

Figure 5 Screenshot of the simulator running a simple Hello World program with the debugger (see Appendix B for source)

[13] One interesting feature supported by Z80Sim was the ability to parse debug information files produced by some C compilers and then use the information contained therein to provide the user with additional information (e.g. what function is currently being executed). While this feature is useful when developing new software, it does not help when analyzing software without access to a debug information file or to the source, as is the case when reverse-engineering.

to parse a new instruction. The interpreter type-checks the parse tree produced by the front-end and then performs the operations specified by said tree.

The resulting debugger looks and acts much like the common open source command-line debugger gdb, and supports most of the same features as gdb: breakpoints, variables, a disassembler, access to processor registers and memory, and basic arithmetic and logical operations. We also added specialized instructions for the Z80 and for the Advantage's peripherals. These instructions include mechanisms to simulate pushes of buttons on the Advantage's peripherals, to display the output of various peripherals (LEDs and LCDs), to set breakpoints on port IO instructions, and to perform port IO instructions. For a full list of the commands the debugger currently supports see Appendix A.1. The functions described in the interface of 0 (**`Print`**, **`ToggleButton`**, and **`PrintState`**) along with the basic memory read and write functions of the memory module, the port IO functions of the simulator core and a set of functions to examine the emulated Z80's state provide all of the mechanisms used by the debugger to interface with the simulator peripherals.]

4.5 Simulator Core

```
main()
   parse arguments
   load configuration file
   for each peripheral module
      module_Initialize()
   until stopped
      if debugging
      then
         let s = line read from debugger
         lex, parse and interpret s
      else
         step_processor
         if a breakpoint is encountered then set debugging
      for each peripheral module which requires a push of timing information
         module_Tick()
      for each peripheral that has a user interface
         module_UpdateFrontEnd()
      transmit all queued front-end messages to the front-end
      poll front-end for input update messages
      for each message from the front-end
         let module be the component the message refers to
         module_ProcessMessage(message)
   for each peripheral module
      module_finalize()
```

Algorithm 1 Pseudocode for the main function of the simulator core

Most of the functionality of the simulator core is encapsulated in the main function described by Algorithm 1. In addition to this main function, the simulator core also contains functions to handle the demultiplexing of port IO requests. These additional functions are exposed to the processor and provide the interface between the processor and its many peripherals. As a part of this interface, the simulator core has access to the processor's data and address buses as specified in the basic architecture diagram (figure 3).

5 Results

The simulator we produced in this project implements a simulation of all the peripherals that have been reverse-engineered at the time of the writing of this paper. This set of peripherals includes the memory management subsystems, the voter and operator panels, the real-time clock, and the power controller, to name a few. The simulator can now be used to run the Advantage's firmware and examine the Advantage's state during execution. In timing tests, the simulator has performed reasonably well, approaching real-time performance with speeds of about 4 MHz when run without the front-end.

The debugger we produced provides a number of features geared specifically towards reverse engineering the Advantage's firmware and both the simulator and debugger already have been of use in analyzing the firmware. Thus far, we have used it to determine the sequence of code executed during the Advantage's boot up process – a non-trivial task using just static code analysis. The simulator also has proven useful for identifying connections between sections of code previously thought to be unrelated. The debugger's ability to break on port IO instructions accessing a specified port has also been helpful in determining what sections of the firmware are responsible for interacting with each peripheral – another task which is difficult (if not impossible) with static code analysis. As the analysis of the hardware and firmware proceeds, we expect the simulator to be valuable for working out missing details and identifying errors in our understanding of the hardware (to date, the simulator and debugger have not been particularly useful for reverse-engineering the hardware) [8]. The simulator and debugger can also be used to develop and test new firmware and applications for the Advantage – Appendix B contains the source for a simple Hello World program that runs in the simulator and makes use of the Advantage's operator panel – and so it is now possible to write vote stealing software and to initially test and debug it in simulation before running it on an Advantage.

One current shortcoming of the simulator is that when executing the Advantage's firmware, the simulator eventually halts because portions of the Advantage's firmware are only executed in response to raised interrupts and the simulator does not yet implement the recently reverse-engineered interrupt controller.

There are three obvious areas in which future work on the simulator can be directed. The first area is the back-end of the simulator. The reverse-engineering of the Advantage is still a work in progress, and, as such, at the time of writing, some of the peripherals are still being reverse-engineered. One obvious task is to implement in the simulator the peripherals that have recently been reverse-engineered or will be reverse-engineered in the future. By implementing these additional peripherals, the simulator should then be able to successfully run the original firmware of the Advantage. With the ability to run the firmware, it would then be possible to fully utilize the specialized features of the debugger to aid in analyzing the Advantage's firmware and in discovering possible attack vectors. Another issue with the back-end is that while it achieves reasonable speeds, it should be possible to speed up the emulation a fair amount by switching to a more table-driven representation of the processor and by cleaning up peripheral modules.

The second area is the simulator's front-end. Currently, the front-end uses the standard Swing look-and-feel for buttons and rectangles for LCDs and LEDs. One improvement would be to modify the GUI so that it more closely resembles the interface of the actual Advantage. Another improvement to the front-end would be to add to the GUI some of the more recently reversed-engineered components that require user interfaces to the GUI. Also, because the front-end communicates with the simulator through sockets, it should be possible to modify the front-end

such that it could be attached to a simulator running on a different computer, paving the way for users to experiment with the simulator and the Advantage's firmware, without ever actually having access to the source of the simulator or the Advantage's firmware. Also, the interface between the simulator and the front-end is extraordinarily slow and when run with the front-end, the simulator falls far short of its near real-time performance; another improvement would be to modify this interface so that it is not so much of a bottleneck.

The third area is the debugger. Recently, the reverse-engineering team has determined that the original firmware was written in C and compiled for the Z80. As such, it is likely that some sort of calling convention is followed in the binaries. A common tool in many debuggers – which is particularly useful for reverse engineering – is to examine stack frames to ascertain the sequence of function calls made to reach any point in the execution of a program without having to step through every instruction. With knowledge of the calling convention, it could be possible to add a similar tool for examining stack frames to the debugger which would aid in the analysis of the firmware.

Acknowledgements

This paper and project would not have happened without the help and support of a number of people. Professor Andrew Appel led this project, initially securing the voting machines and allowing me to become involved; he also provided indispensable advice about the basics of the simulator and provided many useful suggestions for features to add to aid in the analysis of the Advantage. Alex Halderman reverse-engineered the Advantage's hardware and wrote the functional specifications which were the basis for this simulator; he also was always willing to answer questions and clarify details about the behavior of the Advantage. A portion of the code used in this project was borrowed from other open source projects and credit is due to Lorenzo Lucchini for his Z80 emulator and to Ge Wang and Peng Bi for their cross-platform socket library. Also, many thanks to my proofreaders – without their valuable feedback this paper would not be remotely readable.

References

[1] Andrew W. Appel. Affidavit in the case of Gusciora v. McGreevey, N.J. Superior Court. http://www.cs.princeton.edu/~appel/nj-voting-case/Appel-feb07-certif.pdf, February 2007.

[2] Andrew W. Appel. How to defeat Rivest's ThreeBallot Voting System. http://www.cs.princeton.edu/~appel/papers/DefeatingThreeBallot.pdf, October 2006.

[3] Robert J. Boram. "Electronic Voting Machine and System." US Patent 4641240. February 3, 1987.

[4] Tracy Campbell. *Deliver the Vote: A History of Election Fraud, an American Political Tradition – 1742-2004*. Carroll and Graf, 2005.

[5] Election Data Services. 2006 Voting Equipment Summary By Type as of: 11/07/2006. http://www.edssurvey.com/images/File/VotingEquipStudies%20/ve2006_report.pdf, September 2006.

[6] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security Analysis of the Diebold AccuVote-TS Voting Machine. http://itpolicy.princeton.edu/voting, September 2006.

[7] J. Alex Halderman. AVC Advantage: Hardware Functional Specifications. May 2007

[8] J. Alex Halderman. e-mail and Google Talk communication, May 2007.

[9] Harry Harrison. *The Stainless Steel Rat for President*. Bantam Books, 1982.

[10] Richard Hofstadter. *The American Political Tradition: And the Men Who Made It*. Alfred A. Knopf, 1948.

[11] Robert F. Kennedy, Jr. "Was the 2004 Election Stolen?" *Rolling Stone*. http://www.rollingstone.com/news/story/10432334/was_the_2004_election_stolen, June 2006.

[12] Lorenzo J. Lucchini. "Z80Sim." March 2005. *Sourceforge.net*. May 2005. http://sourceforge.net/projects/z80sim.

[13] *Man of the Year*. Dir. Barry Levinson. Universal, 2006.

[14] Sam Michaels. "GameBoy Emulators." *Zophar's Domain*. http://zophar.net/gb.html.

[15] Ronald L. Rivest. The ThreeBallot Voting System. http://theory.csail.mit.edu/~rivest/Rivest-TheThreeBallotVotingSystem.pdf, September 2006.

[16] "Sideshow Bob Roberts." *The Simpsons*. FOX Broadcasting Company. KTVR, San Francisco. 9 October 1994.

[17] Zilog Corporation. Z80 Family CPU User Manual. http://www.zilog.com/docks/z80/um0080.pdf, March 2005.

A Simulator Documentation

A.1 Simulator Debugger

The Simulator debugger is a debugger that supports memory access, updates and breakpoints, it is loosely modeled on the behavior of **gdb**.

A.1.1 Value Definitions

$(*PC|SP|IX|IY|HL|AF|BC|DE|HL'|AF'|BC'|DE'|A|B|C|D|E|F|H|L|A'|B'|C'|D'|E'|F'|H'|L'*)
Accesses one of the registers; ' is used to access the shadow registers. Register access is case-insensitive. This is an integer type.

0(x|X)#
Denotes a hexadecimal number. This is an integer type.

#
Denotes a decimal number. This is an integer type.

V:#
Used to define a 'virtual' address; by virtual I mean that this address acts like an address put on the Advantage's address bus, accessing RAM or ROM chips as determined by the current settings of the data or program memory subsystems. This is an address type.

D:#,#
Used to define an absolute data address. The first # denotes a block (a chip and a base address in that chip), and the second # an offset (up to 0xffff) in that block. This is an address type.

Table 5. Data Addressing Blocks

Block #	*Chip/Base Address*
0	128KB SRAM1: 0x0
1	128KB SRAM1: 0x10000
2*	128KB SRAM2: 0x0
3*	128KB SRAM2: 0x10000
4	32KB SRAM: 0x0

***If the RAM_Jumper is not set in the config file, attempts to access block 2 or 3 will fail**

P:#,#
Used to define an absolute program address. The first # denotes a block (a chip and a base address in that chip), and the second # an offset (up to 0xffff) in that block. This is an address type.

Table 6. Program Addressing Blocks

Block #	*Chip/Base Address*
0	EPROM1:0x0
1	EPROM2:0x0
2	EPROM3:0x0

S:#
Used to define an address on the serial ROM chip. The # denotes the address on the chip (up to 0xff). This is an address type.

X:*#*
Used to define an address on the volatile (non-battery backed) 8KB RAM chip. The # denotes the address on the chip (up to 0x1fff). This is an address type.

A.1.2 Operations

Most of the standard C unary and binary operators are implemented. In particular the binary operators `*,/,%,<=,<,>,>=,==,!=,!,~,&&,&,||,|,^` all work with integral types (not with address types). Note that *v = 5 does NOT act like it would in C. To write to a memory address use the **:=** operator.

v1 **+** *v2*
Can add integer types to create another integer, or can add an integer type and an address to create an address.

v1 **-** *v2*
Can subtract integer types to create another integer, or can subtract an integer type from an address to create an address.

v
Dereferences an address type and returns an integer type. If *v* is an integer type, it is coerced to a virtual address.

*v1***[***v2***]**
Equivalent to *(*v1* **+** *v2*). If *v1* is an integer type, it is coerced to a virtual address. *v2* must be an integer type. Returns an integer type.

v1 **=** *v2*
Assigns a value to *v1* if it is a variable (not a breakpoint variable) or a register. If *v1* is a register and *v2* is an address: assigns the address' offset to the register. Otherwise assigns *v2* to *v1*.

v1 **:=** *v2*
Writes a byte to the memory address specified by *v1*. If *v1* is an integer type, this assumes *v1* is a virtual address. If *v2* is a 16bit register this operation will fail.

A.1.3 Keyword Definitions

All keywords and aliases are case insensitive. Generating a SIGINT (pushing ctrl-c) while code is executing causes an immediate break.

A.1.3.1 Working with Breakpoints

b *addr*
break *addr*
Used to define and set a breakpoint. If *addr* is a virtual address, the breakpoint applies to that virtual address, not the physical address it refers to. If *addr* is a absolute address (data or program), the breakpoint applies to that absolute address. After execution prints the name of the breakpoint and address.

breakport *port*

Used to define and set a breakpoint on an IO port. Prior to executing an IO In or Out that accesses *port*, this will halt the program and enter debugger mode. After execution prints the name of the breakpoint and port number.

clear *breakpoint*
Used to disable and remove a breakpoint. *breakpoint* must be a breakpoint variable.

enable *breakpoint*
Used to enable an existing breakpoint *breakpoint*.

disable *breakpoint*
Used to disable an existing breakpoint *breakpoint*.

A.1.3.2 Running code

s
step
Single step. Prints the next instruction to be executed.

c *addr*
continue *addr*
Continues execution until a breakpoint is hit or a user-interrupt.

exit
Exits simulator

A.1.3.3 Examining State

dasm *addr*
Attempts to disassemble the instruction starting at virtual address *addr* (program or data addresses are not supported).

p *value*
print *value*
Attempts to print *value*, all numbers are printed in hex (this includes numbers in addresses that are printed).

printop
Prints an ascii representation of the operator panel. Activated LEDs are indicated by an *.

printstate
Prints an ascii representation of the internal state of the simulator peripherals.

printvp
Prints an ascii representation of the voter panel. Activated LEDs are indicated by an *.

printvplcd
Prints an ascii representation of the voter panel LCD screen.

A.1.3.4 Toggling Buttons

oppanel *value*
Toggle operator panel button specified by *value*. 1-12 specify the buttons with numbers 1-12, 13 refers to the Start/Yes button, 14 refers to the Next/No button.

vp *value*
Toggle voter panel button specified by *value*. 1-504 specify one of the 504 buttons on the vote panel. Let i be an integer in the range [0,11], j be an integer in the range [0,5], and k be an integer in the range [0,6], then $i*42+7*j+k$ specifies the button in the i^{th} subpanel, j^{th} column, and k^{th} row (subpanels are numbered from the bottom-left to the top-left, and then bottom-right to top-right, where columns are numbered left to right, and rows from top to bottom); 505 refers to the cast vote button.

vpkeyboard *value*
Toggle voter panel keyboard button specified by *value*.

Table 7. Keypad key mapping

Value	*Button*	*Value*	*Button*	*Value*	*Button*	*Value*	*Button*	*Value*	*Button*
1	G	8	,	15	H	22	-	29	T
2	F	9	.	16	←	23	Z	30	S
3	E	10	M	17	→	24	Y	31	R
4	D	11	L	18	Enter	25	X	32	Q
5	C	12	K	19	Up	26	W	33	P
6	B	13	J	20	Down	27	V	34	O
7	A	14	I	21	’	28	U	35	N

A.1.3.5 Controlling and monitoring processor I/O

in *port*
Returns a byte read from *port* (à la Z80 IN instruction). Note: *port* can be a full word, it is placed on the address bus (low byte in the low 8 bits, high bit in the high 8 bits) like a true IO instruction.

out *port, value*
Writes *value* to *port* (à la Z80 OUT instruction). Note: *port* can be a full word, it is placed on the address bus (low byte in the low 8 bits, high bit in the high 8 bits) like a true IO instruction.

traceport
Toggles traceport mode. When activated, will generate useful information during execution of any IO instruction. Prints the state of traceport after execution.

A.2 Simulator Configuration File Format

The Simulator configuration file is a tab-delimited ASCII file of the following form:

```
Keyword  value
Keyword  value
...
```

Any line that doesn’t start with a reserved keyword is ignored. No line should have length greater than 1024 characters.

A.2.1 Required keywords

P_EPROM_1 *filename*
This keyword is used to specify the file that contains the binaries loaded on EPROM 1.
P_EPROM_2 *filename*
This keyword is used to specify the file that contains the binaries loaded on EPROM 2.
P_EPROM_3 *filename*
This keyword is used to specify the file that contains the binaries loaded on EPROM 3.
Serial_ROM *filename*
This keyword is used to specify the file that contains the data loaded on the configuration EPROM (serial number, key, etc).

A.2.2 Optional keywords

D_32SRAM *filename*
Including this keyword will load 32KB from *filename* into the 32KB RAM chip.
D_128SRAM_1 *filename*
Including this keyword will load 128KB from *filename* into the 128KB SRAM 1
D_128SRAM_2 *filename*
Including this keyword will load 128KB from *filename* into the 128KB SRAM 2
Input_Script *filename*
This keyword specifies a file to be used as an input script. An input script is an ASCII file which specifies a list of bytes (in hexadecimal). Whenever an input instruction is executed for a port which isn't yet supported in the simulator the simulator will attempt to read a line from the input script, grab a byte and return that to the processor. If no lines are left in the input script or no input script is specified the simulator will print a line to stderr specifying the port it attemped to read from.
Output_Log *filename*
This keyword specifies a file to be used for logging all output instructions executed on ports that the simulator doesn't support yet. If not output log is specified, these messages are logged to stderr.
RAM_Persist *{TRUE|FALSE}*
If any of the RAMs are loaded from a file and this is set to true, upon exiting the simulator will dump the final contents of the RAM back to the file from which it was loaded.
RAM_Jumper *{TRUE|FALSE}*
Setting this to true will be equivilant to including the second 128KB RAM chip.
RT_Clock *filename*
Including this keyword will load 128B from *filename* into the realtime clock's RAM.
Results_Cartridge_A *filename*
Including this keyword indicates that a memory cartridge is loaded in slot A. If this keyword is present, the keyword **Cartridge_A_Rev** is requried. This keyword will also cause the cartridge in slot A to be loaded with data from *filename* (the amount read is based on the cartridge's revision).
Cartridge_A_Rev *{REV_C}*
This keyword specifies the revision of the memory cartridge in slot A.
Results_Cartridge_B *filename*
Including this keyword indicates that a memory cartridge is loaded in slot B. If this keyword is present, the keyword **Cartridge_B_Rev** is requried. This keyword will also cause the cartridge in slot B to be loaded with data from *filename* (the amount read is based on the cartridge's revision).
Cartridge_B_Rev *{REV_C}*
This keyword specifies the revision of the memory cartridge in slot B.
FrontEnd *{TRUE|FALSE}*

If this keyword is set to true, the simulator will attempt to connect to the Java front-end over a socket.

B Sample Hello World assembly

```
          ld SP, 0xFFFF
          call SPRINT
String:   defm "HELLO WORLD"
          defb 0xFF           ; sentinal value for hello world string
          ret
SPRINT:                       ; print the string pointed to by the address
                              ; pushed on top of the stack and HL and
                              ; terminated by 0xFF
          pop HL
          ld A,(HL)
          inc HL
          push HL
          cp 0xFF
          jr Z, RESTART
          call PRINT
          jr SPRINT
RESTART:
          ld A, 0x41
          out (0x16), A       ; set LEDs on operator panel
          in A, (0)           ; attempt to read a byte from an unused port
          out (0), A          ; attempt to load a byte to an unused port
          in A, (0)           ; repeat
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          in A, (0)
          out (0), A
          halt
PRINT:    ld C, 0x11
          in B, (C)           ; poll busy bit of the operator panel LCD
          bit 7, B
          jr NZ, PRINT        ; if busy bit is set, spin
          out (0x12), A       ; write character to operator panel LCD
```